Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e Multimédia

Inteligência Artificial e Sistemas Autónomos

Docente: Eng.º Paulo Vieira

Semestre de Verão 21/22

Relatório Final

Relatório realizado por:

Alice Fernandes 45741 LEIM41N

Lisboa, 6 de julho de 2022

Índice de matérias

Índice de figuras

Introdução

No decurso do semestre aprendemos várias coisas sobre inteligência artificial e raciocínio automático. Todo esse conhecimento foi posto em prática com a implementação de uma parte do projeto sobre o tema que foi lecionado. No total foram 12 entregas onde tivemos de pegar no suporte aos trabalhos práticos na linguagem UML que foram disponibilizados durante as semanas e implementar partes dos diagramas UML.

O projeto consiste numa simulação de um ambiente em que um agente tem de encontrar os vários alvos. O ambiente onde o agente está tem obstáculos e tem de conseguir contornar os obstáculos e encontrar o caminho ideal para cada alvo. Para navegar no ambiente, este agente utiliza os vários tipos de "controlo" foram lecionados durante o semestre: reativo, através de procura em espaço de estados, através de processos de decisão de Markov e através de aprendizagem por reforço. O objetivo de aprender essas componentes acima mencionadas é porque a tomada de decisão do agente varia de acordo com a maneira que é definido (se é deliberativo, se utiliza o algoritmo Q-Learning…)

Para o projeto foi disponibilizada uma biblioteca, a biblioteca ECR, que implementa a componente visual, ficando encarregue ao aluno (durante o semestre) a implementação dos vários tipos de controlo acima mencionados.

Nos vários tópicos a seguir vou referir vários métodos de decisão, no sentido em que o agente avalia o ambiente e a sua ação é ditada pelo mecanismo ou método de procura que estamos a utilizar durante a simulação.

Enquadramento Teórico

1. Agentes Inteligentes

Um agente é um é mecanismo que recolhe informação (através de sensores), e realiza ações através de atuadores (ou operadores) num determinado contexto (ou ambiente). Estes agentes podem ser controlados de várias maneiras, e inclusive podem aprender (e memorizar) a navegar pelo ambiente.

Pode também ser racional, no sentido em que, através da informação recolhida através de sensores ou por aprendizagem, consegue perceber qual a melhor ação a concretizar com o objetivo de maximizar os seus ganhos. Para ser racional então ter então algum conhecimento do ambiente, quais a ações que pode tomar, que ações já executou e como medir o seu próprio desempenho.

O comportamento de um agente é definido pela sua arquitetura e "controlo" que utiliza.

Um agente também tem uma propriedade de estado que é uma espécie de memória atual da condição do agente, ou seja, o estado em termos práticos pode onde está, ou o que está a fazer, qualquer outro valor que indique o algo sobre o "presente" do agente

Ambientes

O ambiente é o contexto onde o agente está inserido, e as características do ambiente afetam as soluções encontradas para os problemas. As características do ambiente são as seguintes

- Pode ser totalmente ou parcialmente observável, ou seja, se o agente consegue percecionar todo o que precisa então é totalmente observável, ou se faltar informação por diversos motivos é parcialmente observável
- Pode ter apenas um agente ou múltiplos, e no caso de existir múltiplos agentes é importante perceber como interagem uns com os outros, se colaboram para melhorar a performance cada um ou se competem entre si
- É determinista se o próximo estado do agente for obtido através do estado atual e da ação atual do agente, ou estocástico se não depender apenas das ações concretizadas do agente

- É episódico se a experiência do agente é dividida por episódios (ou execuções). Cada episódio não depende dos episódios anteriores.
- Estático se não se altera enquanto o agente está a deliberar ou dinâmico caso se altere com o tempo
- Discreto se temos um número finito de estados e ações ou continuo se o número de estados é "infinito"
- Conhecido ou desconhecido se o agente conhece o estado seguinte ou a probabilidade do estado seguinte.
-

Arquitetura Reativa

Em síntese, um agente reativo é um tipo de agente que apenas reage ao ambiente. Este tipo de agente perceciona o que está à sua volta e reage mediante os estímulos que as perceções introduzem.

Este tipo de agente não tem conhecimento do ambiente (ou estado do mundo) onde está nem sabe quais são os seus objetivos, e por isso parece que anda "à deriva" até realizar uma ação que o faz atingir um objetivo. Um exemplo dado em aula foram os veículos de Braiternberg, que utiliza sensores de luz para guiar o carro, ou seja, reage de forma direta com o ambiente. O objetivo deste veículo é de encontrar o caminho através da luz, mas o veículo não sabe desse objetivo.

A arquitetura de um agente reativo na sua forma mais simples pode então ser definida como na imagem abaixo

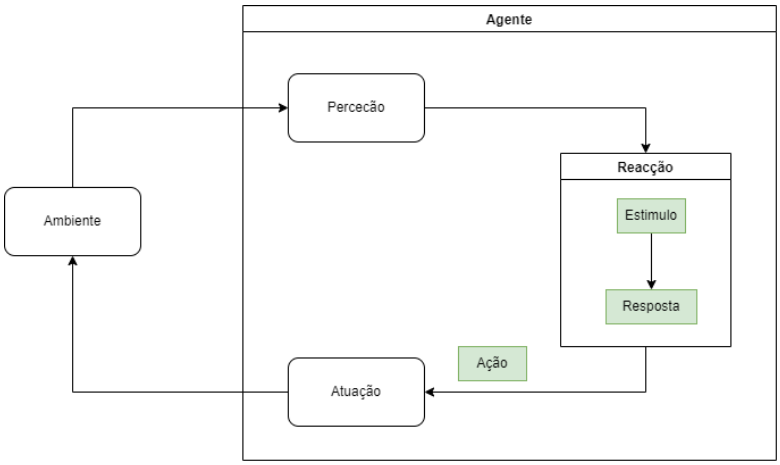

Figura 1 - Arquitetura Reativa - Simples

A imagem anteriormente apresentada não têm em conta que um agente reativo pode reagir de maneiras diferentes para o mesmo estímulo, como por exemplo, se o agente colide com um obstáculo pode se desviar para qualquer outra direção, ou até pode ter várias reações para um único estímulo. Em situações em que é necessário despoletar reações em simultâneo, podemos agrupá-las em comportamentos, ou seja, podemos juntar reações que estão relacionadas num grupo, e agente ao receber um estímulo pode executar um conjunto de ações que está relacionado com um determinado estímulo.

Podemos ter ainda outra camada de abstração, em que comportamentos podem ser compostos por outros comportamentos e ter o agente a reagir com esses comportamentos compostos.

Na componente prática conseguimos simular este tipo de agente, e o que se observa é que de facto parece que o agente está a deriva sem um propósito claro, e vai explorando o mundo onde está e no processo recolhe alguns alvos. Isto pode ser o que é necessário em algumas aplicações, mas se queremos que o nosso agente atinja objetivos, então este tipo de arquitetura não vai ser mais adequada.

Esta arquitetura é utilizada por exemplo em aspiradores inteligentes, que vão aspirando o chão enquanto navegam pela casa. Para a tarefa que estão a realizar não precisamos de indicar objetivos, apenas precisamos que se encontrarem lixo que aspirem e que continuem a explorar a procura de mais.

Arquitetura Deliberativa

Um agente com uma arquitetura deliberativa não reage de forma imediata ao que perceciona, em um processo de decisão mais racional que a arquitetura reativa. Se a arquitetura reativa reagia às perceções, uma arquitetura reativa vai deliberar sobre aquilo que perceciona, e só depois disso vai planear as ações que vai executar para atingir um objetivo. Este tipo de agente mantém um estado interno do ambiente (ou mundo), que depois utiliza para deliberar e selecionar as ações que lhe permitem resolver um problema

De um modo geral, a tomada de decisão segue os seguintes passos:

1. Observar (ou percecionar) o mundo e atualizar o modelo interno do mundo
2. "Reconsidera" e valida se modelo interno foi alterado, e se foi, então delibera e planea uma nova sequencia de ações
    a. Durante a deliberação define qual será o novo estado-objetivo que quer atingir

   b. O planeamento é o "cálculo" da lista de ações que permitem atingir o estado-objetivo definido acima
3. Executa o plano definido ao executar todas as ações do plano.

Dependendo do tipo de controlo pode ter alguns passos intermédios, como por exemplo, nos Processos de Decisão de Markov avalia se um estado é ou não útil, ou na Aprendizagem Automática em que aprende através da exploração qual o melhor conjunto de ações a efetuar.

2. Procura em Espaço de Estados

Uma forma de controlar as decisões do agente é através de uma procura em espaço de estados Este método de decisão já nos permite identificar objetivos explícitos. A procura em espaço de estados permite-nos estruturar um problema de modo indicar o objetivo que queremos atingir e uma estrutura de nós que indica ao agente por onde pode ir para atingir o seu objetivo. O caminho percorrido é a solução do problema.

Este tipo de procura pode ser visualizado em forma de árvore, em que cada nó da árvore está associado a um estado, cada ramo corresponde a uma ação que o agente pode tomar e um nó inicial que representa o estado inicial do agente. O espaço de estados são todos os estados possíveis do ambiente.

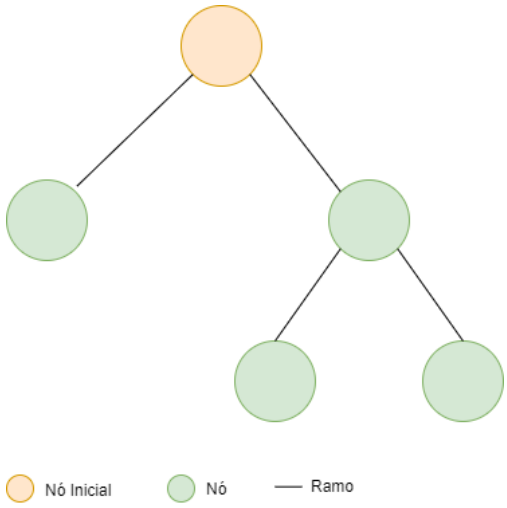

Figura 2 - Exemplo de estrutura em Árvore

Cada transição (ou visualmente, cada ramo da árvore) que agente concretiza tem um custo, que é definido no problema, então o agente tem de conseguir calcular uma solução que minimize o custo, mas que consiga chegar ao estado (ou objetivo) final.

Para além do estado, cada nó da árvore tem outras propriedades associadas: o nó anterior, os seus nós "filhos", o custo para chegar a aquele nó.

Algoritmo de Procura

O espaço de estados é dividido em três partes:

- Os nós que já foram visitados na árvore - Os nós explorados
- Os nós que vão ser explorados – A fronteira de exploração
- Os nós que ainda não foram explorados – Os nós por explorar

Durante a execução do algoritmo, os eles vão ser movidos entre estas três regiões, e fica a cargo da estratégia de procura que é utilizada para resolver o problema e que algoritmo utilizar para a fronteira de exploração. De um modo geral, o algoritmo é o seguinte:

1. Expande-se o nó raiz, e os nós resultantes são acrescentados à fronteira de exploração
2. Dependendo da estratégia de procura, os nós da fronteira são retirados e expandidos, que gera ainda mais nós que são depois acrescentados à fronteira. Ao serem explorados são movidos para a lista de nós explorados.
3. Repetir os passos acima até atingir o nó com o estado-objetivo

A ordenação da fronteira fica a cargo da estratégia de procura. As estratégias de procura podem ser divididas em duas categorias: procura informada, onde temos a informação sobre o objetivo e o domínio do problema, e procura não informada onde não existe informação sobre o domínio do problema, e podem ser avaliadas pelos seguintes parâmetros

- É completa se for possível encontrar uma solução
- É ótima se for a melhor solução
- Pode ser avaliada pelo número de nós que necessita de colocar na fronteira (complexidade espacial)
- Pode ser avaliada pelo tempo que demora a realizar a pesquisa (complexidade temporal)

Procura não informada

Existem quatro estratégias de procura não informada: Procura em Largura, Procura em Profundidade, Procura em Profundidade Iterativa e Procura Custo-Uniforme

Na Procura em Largura, o nós mais superficiais são explorados primeiro, e só passa para o nível de profundidade seguinte quando explora todos os nós no nível de profundidade atual. Para a fronteira, esta procura utiliza uma lista FIFO (First In First Out) na ordenação da fronteira, ou seja, os nós mais antigos são explorados primeiro

A Procura em Profundidade explora os nós mais profundos e só quando chega ao fim da árvore volta ao nó mais superficial e repete o processo. Esta procura utiliza uma fronteira LIFO (Last In First Out) na ordenação da fronteira, ou seja, os nós recentes são explorados primeiro

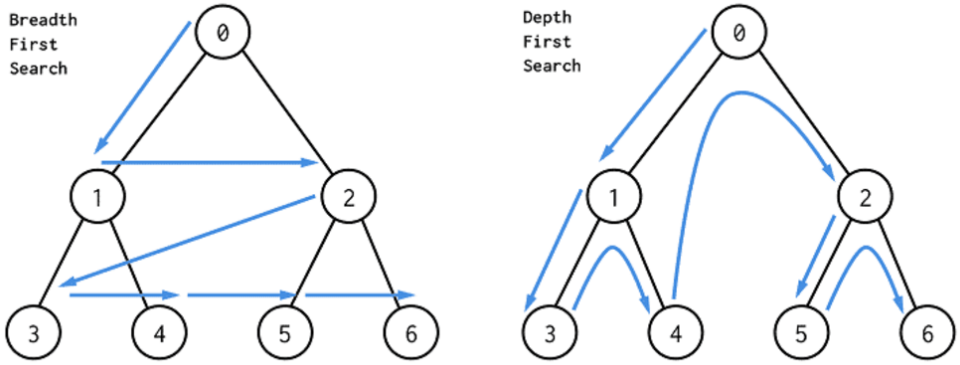

Figura 3 - Procura em Largura e Profundidade[1]

Procura em Profundidade Iterativa segue o mesmo raciocínio que a Procura em Profundidade, mas esta tem um limite de profundidade até onde pode chegar. Após chegar à profundidade limite a procura para de encontrar uma solução.

Finalmente, a Procura Custo-Uniforme é uma extensão da Procura em Largura, mas o custo de cada transição é tido em conta quando queremos explorar um nó. A prioridade é dada aos nós que custam menos, ou seja, a fronteira é ordenada em função do custo que cada nó tem

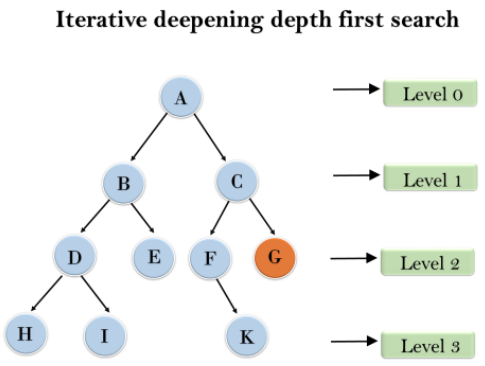

Figura 4 - Procura em profundidade Iterativa[2]

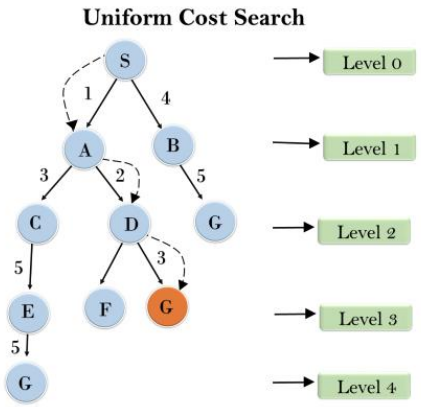

Figura 5 - Procura Custo-Uniforme[3]

[1] Imagem retirada de https://dev.to/danimal92/difference-between-depth-first-search-and-breadth-first-search-6om
[2] Imagem retirada de https://www.javatpoint.com/ai-uninformed-search-algorithms
[3] Imagem retirada de https://www.javatpoint.com/ai-uninformed-search-algorithms

Em termos de comparação, as procuras em largura têm armazena muitos nós na fronteira, ou seja, tem uma complexidade espacial exponencial, mas temos a garantia que obtemos a melhor solução. A procura em profundidade tem uma complexidade espacial menor que a procura em largura, mas nem sempre encontra a melhor solução ou pode não encontrar solução de todo.

A procura custo-uniforme consegue encontrar a melhor solução, mas pode ter uma grande complexidade espacial e temporal que as outras procuras. A procura em profundidade iterativa retira os benefícios da procura em largura e profundidade e é capaz de encontrar a solução ótima

Procura informada

Existem duas estratégias de procura informada: Procura Sôfrega e Procura A*. Ambas as procuras utilizam uma função de heurística que permite estimar o custo da transição entre nós. Esta função pode ter qualquer implementação (como por exemplo distâncias), basta apenas que retorne o custo, mas não pode estimar este custo em excesso (tem de ser otimista). Idealmente o custo estimado é menor ou igual ao custo real de expandir os nós, ou seja, é admissível se não sobrestimar o custo. Esta heurística determina na prática a organização da fronteira de exploração. Utiliza-se esta função para estimar por exemplo a distância de um nó até ao nó objetivo. Utilizam também uma função de avaliação que pode têm em conta a heurística.

A Procura Sôfrega expande o nó com a heurística mais pequena. A sua função de avaliação é a função de heurística.

A Procura A* utiliza o custo do nó mais a heurística para avaliar qual o nó a expandir e expande o que minimiza esse custo.

3. Processos de Decisão de Markov

Processos de Decisão Sequenciais

Os Processos de Decisão de Markov (PDM) é uma maneira de tomar decisões sequencialmente, ou seja, tomar decisões através das decisões já tomadas. Neste tipo de processos de decisão não é possível reverter para o estado anterior, por isso as transições de estado são tomadas de forma atingir o objetivo do problema, e só ao fim de algumas decisões é que conseguimos atingir uma recompensa, que pode ser neutra, positiva ou negativa. A utilidade de uma ação depende das decisões tomadas, e as decisões são tomadas sem saber se ajuda ou não a atingir o objetivo.

Para cada estado vamos então ter um conjunto de ações que podem ser executadas, cada uma com uma probabilidade, isto porque o ambiente é não determinista e cada transição pode levar a uma recompensa positiva ou negativa. O próximo estado é determinado com base num valor de utilidade, que tem em conta vários fatores, e que diz qual é a melhor transição (de estado) possível num determinado estado.

Utilidade

Concretamente, através do PDM, é possível calcular um valor de utilidade para cada transição de estado, isto porque processo estocástico tem uma propriedade chamada de Markov, que diz que a probabilidade dos estados futuros depende exclusivamente dos estados atuais.

Para isso, precisamos de representar o mundo em PDM:

- S – Conjunto de estados do mundo
- A(s) - Conjunto de ações possíveis no num estado s pertencente a S
- s’ – Representa o estado seguinte
- a - Representa uma ação que permite ir para um novo estado
- T(s,a,s’) – Probabilidade de transição de s para s’ por a
- R(s,a,s’) – Recompensa prevista na transição T(s,a,s’)
- U(s) - Utilidade para estado s
- γ (Gama) – Fator que representa a passagem do tempo, ou seja, desconta na recompensa para não terem todas o mesmo peso.
- t – Tempo

,
Para determinar a utilidade de cada estado temos de ter em conta vários fatores:

- As recompensas anteriores
- O fator de desconto temporal
- A utilidade dos estados adjacentes
- As possíveis recompensas que é possível obter a partir de um estado, ou seja, a utilidade também é influenciada por um estado s' com uma recompensa negativa que seja possível de atingir a partir de s.

O cálculo da utilidade pode ser feito somando as recompensas e aplicando o fator γ. Podemos então aplicar a fórmula abaixo.

$$U([s_0 \dots s_n) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_n)$$

Política Comportamental

Se cada estado tem uma utilidade então conseguimos descobrir qual a ação possível para conseguir chegar ao objetivo. E é essencialmente isto que a política comportamental é. Representa o comportamento (melhores ações a tomar) que o agente tem de ter para conseguir chegar à recompensa, ou seja, maximiza a utilidade em cada transição de estados.

Solução Ótima

Podemos também calcular a utilidade de um estado através dos estados sucessores através das equações de Bellman, ou seja, a utilidade de um estado é calculada com base na melhor ação possível para esse estado e todos os estados a seguir.

$$U^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} T(s,a,s') \Big[R(s,a,s') + \gamma U^{\pi}(s') \Big]$$

Figura 6 - Equação de Bellman

Ao fazermos isso estamos a decompor este problema em vários sub-problemas, que é a base do princípio da solução ótima que diz precisamente que a utilidade do estado pode ser determinada em função dos estados sucessores.

Sendo que ação executada dá origem a uma política comportamental diferente, precisamos então de obter a política melhor maximiza o valor da utilidade para cada ação possível (ou seja,

a melhor política) a que chamamos política ótima. Podemos simplificar a equação acima, tendo por base uma política ótima π^*

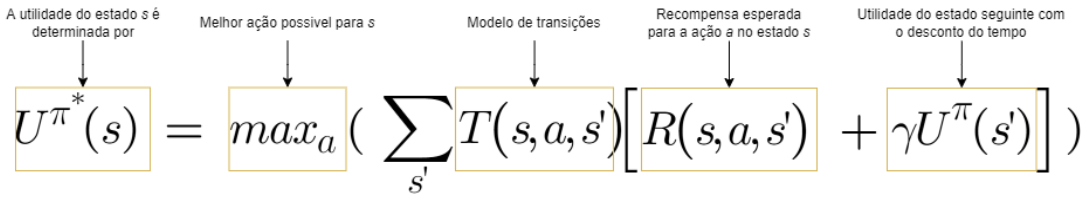

Figura 7 – Cálculo de utilidade - Caso Geral

Se a recompensa só depender do estado que foi atingido então podemos retirar o estado *s'* e a ação *a* do cálculo da recompensa porque já não depende dos estados sucessores, então ficamos com a fórmula abaixo

$$U(s) = R(s) + max_a(\sum_{s'} T(s,a,s') [\gamma U(s')])$$

Figura 8 – Cálculo de Utilidade - Recompensa "fixa"

Algoritmo

O algoritmo para os Processos de Decisão de Markov têm essencialmente três passos:

- Iniciar a utilidade de cada estado a 0
- Para cada estado *s* calcular a utilidade através da fórmula acima. Este passo corre dentro do ciclo que é parado quando a utilidade atinge um limiar.
- Quanto mais cálculos fizermos mais perto ficamos de obter uma política ótima

Quanto mais alto for o fator γ, mais iterações vamos precisar para atingir a política ótima. Podemos sair do ciclo validando se diferença entre as utilidades de cada estado atinge um limiar de convergência.

Quando atinge podemos assumir que o algoritmo convergiu e conseguiu encontrar a política ótima.

Algorithm 1: Value Iteration

Input: Reward function R(s)
Output: Value function V(s)
begin
  $V(s) \longleftarrow 0 \quad \forall s$
  repeat
    forall the s do
      $B(s) \longleftarrow R(s) + \max_a \gamma \sum P(s,a,s')\, V(s')$
    end
    $V(s) \longleftarrow B(s)$
  until *V(s) converges*;
end

Figura 9 – Exemplo de algoritmo PDM em Pseudocódigo[4]

O algoritmo que utilizamos para o projeto têm algumas adições, mas genericamente podemos calcular a utilidade usando o método acima.

[4] Imagem retirada de http://www.scielo.org.mx/scielo.php?script=sci_arttext&pid=S1665-64232016000500300

4. Aprendizagem por Reforço

O agente pode também aprender com o que já sabe do ambiente, ou seja, com o conhecimento disponível, o agente pode escolher a ação que mais se adequa no estado em que está, como por exemplo, desviar-se de obstáculos. A ideia geral é que para treinar o agente por este modo é preciso castigar o agente quando se movimenta (custo de se deslocar) e também recompensar o agente quando atinge um objetivo, e ao deixar o agente explorar o ambiente, o algoritmo vai recolher métricas que lhe permitem tomar melhores decisões.

Genericamente, e pegando nos conceitos já explicados acima, a cada transição de estados disponível associamos um valor de reforço que é na prática o resultado da ação tomada. Este valor pode ser negativo ou positivo, sendo que um valor positivo sinaliza que a ação que executou atingiu um objetivo, e um valor negativo significa que houve um custo em executar uma ação. Este valor é representado como $Q(s,a)$, ou seja, para cada estado s e ação a vamos ter um valor Q associado.

O cálculo do valor $Q(s,a)$ pode ser feito de várias formas:

- Podemos fazer a média com os reforços que vão sendo obtidos com um número de episódios , sendo que este numero é definido pelo programador,

$$Q_n = \frac{r_1^a + r_2^a + \cdots + r_n^a}{n}$$

- Podemos fazê-lo de forma incremental

$$Q_{n(a)} = Q_{n-1}(a) + \frac{1}{n}[r_n^a - Q_{n-1}(a)]$$

- Ou se o ambiente mudar em durante o tempo, podemos calcular o valor de $Q(s,a)$ com base nos valores de $Q(s,a)$ já conhecidos ao logo do tempo (Aprendizagem por diferença temporal)

A aprendizagem por diferença temporal é essencialmente uma maneira de utilizar o que já conhecemos de par estado/ação no passado e com base no novo reforço tentar prever o que será o novo valor de Q para a nova transição. Esta equação introduz um fator de aprendizagem α que valoriza o novo reforço e um fator de "passagem de tempo" γ que desvaloriza a previsão do novo $Q(s,a)$ em função do tempo que já passou.

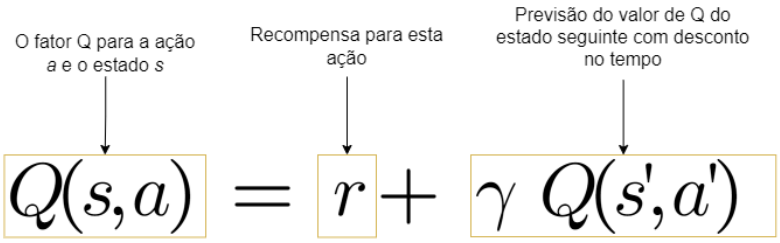

Figura 10 - Aprendizagem por diferença temporal

O valor Q quantifica a aprendizagem que o agente faz, para cada estado/ação. Sendo que a aprendizagem é continua, se só tivermos a explorar então nunca vamos utilizar o conhecimento que já temos sobre o ambiente, então precisamos de algum modo de parar de explorar para conseguirmos aproveitar já sabemos.

Para fazer isso, podemos sempre escolher a ação que maximiza o reforço (ou valor Q) ou através de um número aleatório que determina se ao agente explora, ou se aproveita o que já sabe. A estas estratégias de seleção de ação chama-se Estratégia *Greedy*, e Estratégia ε-*Greedy* respetivamente. Idealmente, escolheríamos sempre a ação que maximiza o valor de *Q(s,a)* porque queremos sempre executar a ação que tem o melhor reforço, ou seja, aqui a politica comportamental ótima é com base no reforço ao invés de utilidade

Estratégia *Greedy*

$$a_t = a_t^* = \max_a (Q(s,a))$$

Estratégia ε-*Greedy*

$$a_t = \begin{cases} a_t^* \\ \text{ação } aleatória, probabilidade\ \varepsilon \end{cases}$$

Existem dois tipos de aprendizagem onde que o que muda é a maneira como a ação é selecionada, ou a política de seleção. A política de seleção única (*On-Policy)* utiliza a mesma estratégia de seleção de ação para o comportamento e para calcular o valor de Q. A política de seleção de aceção diferenciada (*Off-Policy)* utiliza estratégias de seleção de ação para o comportamento e calcular o valor Q

Em termos de algoritmo, existem dois: O algoritmo SARSA e o algoritmo *Q-Learning*. A diferença entre os dois algoritmos é no modo de calcular o valor de *Q(s,a)*. O algoritmo SARSA explora todas as ações e o algoritmo *Q-Learning* utiliza sempre o valor de *Q(s,a)* mais alto

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma Q(S',A') - Q(S,A)\right]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figura 11 - Algoritmo SARSA[5]

[5] Imagem retirada de Sutton, R., & Barto, A. (2017). Reinforcement learning: An introduction

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figura 12 - Algoritmo Q-Learning[6]

[6] Imagem retirada de Sutton, R., & Barto, A. (2017). Reinforcement learning: An introduction

Projeto Realizado

O projeto realizado foi essencialmente a tradução da componente teórica para um programa escrito em *Pytthon*. As arquiteturas deliberativas e reativas são pacotes que podem ser usados para controlar o agente, e recebem o comportamento que utilizam para decidir que ações executar. No trabalho, a arquitetura deliberativa pode ser usada com a procura em espaço de estados e também com os Processos de Decisão de Markov, e o que essencialmente faz é implementar o algoritmo genérico referido no capítulo de arquitetura deliberativa. As estratégias de decisão podem ou não utilizar o mecanismo de procura genérico.

As procuras em si estão definidas num outro pacote e têm todas as procuras que demos no decorrer da cadeira. A ordenação da fronteira de exploração é dada através dos avaliadores que serve cada procura. As heurísticas para as procuras informadas são genéricas, e têm de ser implementadas pelo programador, seguindo as regras acima mencionadas.

Os Processos de Decisão de Markov não utilizam o mecanismo de procura genérico, porque implementam outro tipo de algoritmo para encontrar a solução, tal como o pacote que implementa a Aprendizagem Automática.

No fim, cada estratégia é utilizada num tipo de procura, que pode ser executado a partir do pacote de teste.

Propostas de revisão do projeto realizado

No decorrer do projeto achei as vezes difícil de implementar a componente teórica, e isso se calhar pode-se dever ao facto do projeto ter muitos ficheiros e muitas pastas, e é fácil ficar sem a noção onde fica cada parte do projeto. Um melhoramento seria então talvez de organizar uma estrutura mais simples de recordar, para ser mais fácil de criar um mapa mental da organização do projeto.

Outra coisa que se podia fazer era implementar testes por módulo, para garantir que o comportamento que estamos a ter é o suposto.

Conclusões

Um dos problemas que mais me afetou foi como transformar a componente teórica em prática. Traduzir o conhecimento em código não é uma tarefa fácil e em alguns casos achamos que uma coisa funciona até verificámos que afinal não funciona mais à frente.

Isso aconteceu-me ao implementar a componente de Processos de Decisão de Markov, que ao testar verifiquei que todo o código das heurísticas não estava a ser executado, e precisei de refazer algumas partes do mecanismo de procura para utilizar as funções de memorização das memórias de procura.

A parte do Q-Learning já foi mais fácil de traduzir, até porque era mais simples de compreender e acabámos por ter ajuda do Engenheiro na implementação.

Outra dificuldade que tive foi em compreender os diagramas de UML no sentido em que (em determinados diagramas) difícil de compreender as assinaturas das funções, ou em alguns casos faltavam parâmetros nas assinaturas, que depois só mais a frente é que consegui corrigir. Outra coisa que acho que faltou foi informação um pouco mais concreta nos diagramas, porque as vezes, ao ver um diagrama de classes, não é claro o que se tem de implementar e torna a componente prática confusa.

De resto achei que a matéria mapeava bem para o código e achei particularmente interessante a procura em grafos. A maneira como desenvolvemos a aplicação deixa ao critério do programador o problema que quer avaliar, e facilmente podemos escrever um novo problema e utilizar esta biblioteca para resolver outro tipo de problemas.